Local's Hit- Final Report

DBE14: Distributed Systems

Summer Semester 2020

Group 16:
Frederick Dehner (767417)
Markus Drespling (767873)
David Lüttmann (767907)

LOCALS-HIT

Table of contents

1. Introduction

What is our distributed application about?

1.1 Problem Statement

How to implement a distributed system which meets all the challenges?

We need to develop a heterogenous but open system. Which also has to comply the security requirements of confidentiality, integrity and availability. Our distributed system must be scalable and resistant to faults. It also has to fulfill concurrency and transparency. As you can see from real life examples like the Apple Air Pods, it is a difficult task to fulfill all named requirements while building a distributed system. “A fundamental result in the theory of distributed systems is that under certain conditions, including surprisingly benign failure conditions, it is impossible to guarantee that processes will reach consensus.” [1] In the following report, we will describe how we tried to solve the named problems by implementing our program which is called *local’s hit.*

1.2 Project Description

Now at times of corona virus pandemic and lockdown, it is hard to stay home and avoid social contacts. Especially students have to study at home, can’t see their friends and could not life the typical “student life”. Therefore, to stay connected within your student community we want to build a social media platform, where the user can anonymously share quotes with people.

It is best described with “a snapchat for short quotes (like twitter)”. Our goal is that all users, despite the (social) distance, are close to funny events, interesting finds, or other occurrences. We want the users to forget about their problems and give them a platform to communicate without the barriers of appearance, prejudice, origin, income, and names. This is possible because of the anonymity and avoids racism and discrimination against individual members.

Therefore, we need a front end server through which the primary server communicates with the clients and connects to the back end servers. With several back end servers we provide the shared content on our front end and replicate shared quotes. When entering our program, the client can choose in which region he wants to interact with the community.

At the following chapter the technical requirements will be focused. The decision of the specific characteristics and other possible options will also be described. This report is mainly based on the material from the distributed systems course in the summer semester 2020 including the lectures and the book “Distributed systems: concepts and design” by Coulouris Dollimore and Kindberg in the 5th edition from 2012.

2. Architecture Design

What is the initial idea for the architecture design?

2.1 Architectural Models

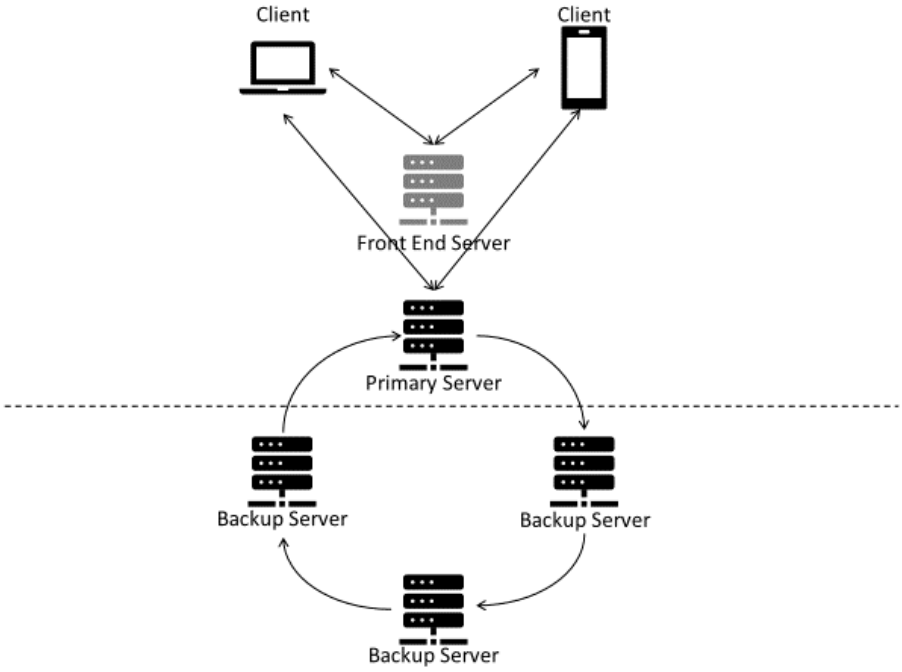


Figure 1: Architectural design

How the distributed system starts working and gets ready for client requests:

1. Back end servers starting, establish distributed system with dynamic discovery, ring and leader election.
2. Leader server starts web service
3. Leader server sends its IP to front end
4. Client asks front end for web page at URL https://[frontend-ip]/index.html
5. Front end responds with web page
6. Client runs javascript code and asks front end for leader at https://[frontend-ip]/
7. Front end responds with leader IP
8. Client establishes tcp socket connection to leader IP
9. If the leader is replaced by another, the previous leader sends a message (CL) to the client to shut down its socket connection.

Message Format:

To ensure that communication runs smoothly between the several servers and also between the primary and the clients, we use prefixes to sort the different types of messages. By the prefixes, each participant of our distributed system has the possibility to immediately assign the messages to a specific task.

In the following table the different prefixes are allocated to the tasks explained earlier.

Type (prefix)	Parameter 1	Parameter 2	Task description
SA	IP address (pid)	-	Service announcement – announce back end service to multicast
RP	IP address (pid)	-	Reply to sender of service announcement with own IP address
SE	Leader candidate IP (pid/mid)	isLeader (True/False)	Start election – election messages in the ring
LE	IP address of current leader	-	Notifies front end server about current leader
HB	Heartbeat GUID	-	Heartbeat messages
FF	IP address of failes node	-	Failure message when no heartbeat is received in the interval
CO	Content	-	Content message – send to clients
CL	Message	-	Notify clients that web socket of backup server is shutting down
CR	Comment	-	Message type for comments on quotes from the client

3. Requirement Analysis

How do we address the project requirements?

3.1 Dynamic discovery of hosts

A necessary precondition to establish any communication is the identification of the participants. The communication initiator also must identify a role participant or group that it will send a message to. But the main question is, how can a new participant find someone when it has no knowledge about available participants? The answer will be described in the following chapter. Also, the implementation of the dynamic discovery of hosts in our project will be elucidated.

The dynamic discovery follows these steps:

1. To receive service announcements from new participants, each recipient continuously has to listen for them.
2. In the first step, a new participant who wants to join a distributed system from which it has no knowledge of any specific recipient but only of a multicast address, sends a service announcement message. This message (SA) includes the address of the new participant. Then the server waits two seconds for a response.
    - SA(pid), with pid = the participants ID which is sent

If there already exists a distributed system:

3. The next step includes the receiving of the service announcement from potential recipients. The recipients update their group views if they receive a new service announcement message.
4. After the group view update each recipient respond with reply messages (RP) in a third step. This reply messages include the recipient's address.
    - RP(pid)
5. Within its response time of two seconds, the new participant collects all replies and creates its own group view. Afterwards to receive service announcements from new participants, it continuously has to listen for them.

If there is no system available yet:

3. The new participant does not get any response in the time interval of two seconds
4. The server goes on with the tasks of the ring formation and the leader election (described in the following) with itself.
5. To receive service announcements from new participants, the server continuously has to listen for them.

The architecture of dynamic discovery of an already existing system:

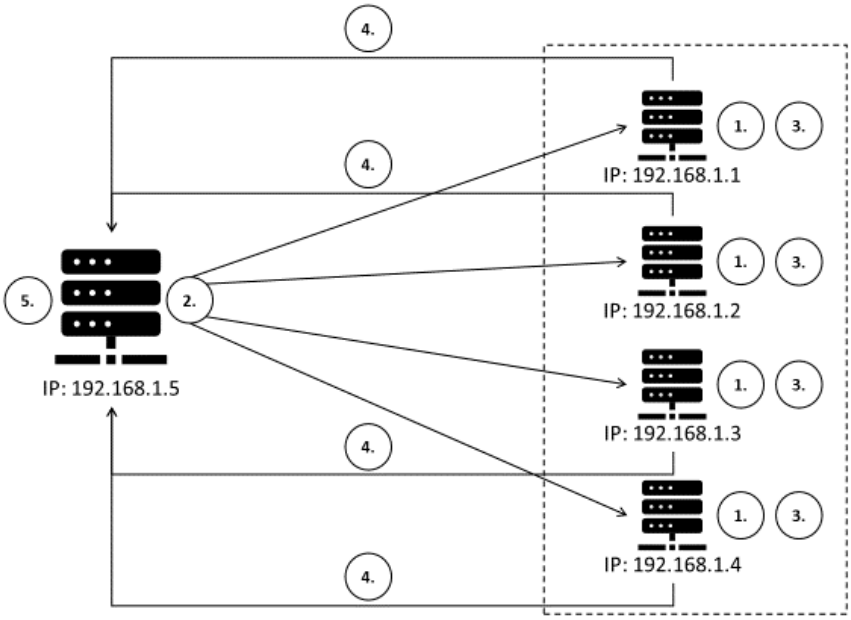


Figure 2: Dynamic discovery of hosts

We will use multicast for service announcement, because we do not know all participants of the distributed system at the beginning. With multicast we limit the number of recipients that only servers who are intended to participate at the distributed system will receive the message.

There are two different types of transport protocols which can be used to send messages through a network. UDP (User Datagram Protocol) and TCP (Transmission Control Protocol) differ in many aspects and are suitable for different purposes.

Because of the connection-free protocol and its fast, ease and efficient delivery with a small header size UDP is more suitable for broadcast and multicast transmissions. Therefore, we will use the UDP protocol in our project. [2] The process will be implemented with sockets. In addition to the servers, which were determined via the dynamic discover of hosts, our system also includes a static front end server, which we added as web server. Details about this can be found in the implementation chapter.

3.2 Voting Algorithm/Leader Election

In this section the leader election will be considered by a voting algorithm. Therefore, the general issue of how to "elect" one participant of a collection of servers to perform a special role will be analyzed. [1] This special role, further referred to as leader, is required to prevent interference and ensure consistency when accessing the resources. For the sake of consistency it is necessary to choose just one server to fulfill that role. [1] Other leader responsibilities, as learned in the lecture of the distributed systems course, are the coordination of activities, modification of data, handle of faults and the role as a sequencer.

It is essential that all the participants agree on the choice. Afterwards, if the leader fails or wishes to retire and several surviving servers can fulfil that role then another leader election is required to choose a replacement. [1] Every server of the system can initiate a run of the election algorithm. But there are two requirements to be fulfilled. An individual server does not call more than one election at a time, but in principle N servers could call N concurrent elections. Secondly, the choice of the elected participant must be unique, even if several participants call election concurrently. The server with the largest identifier will be chosen as the leader. An identifier may be every useful value if the identifiers are unique and totally ordered. [1] In our particular case, we will use the servers IP address as the unique identifier.

Central requirements during any particular run of the algorithm are safety and liveness. Safety means, that a leader is chosen as the non-crashed server at the end of the run with the largest identifier. Liveness refers to the participation of all servers and their eventuality to crash or not to elect. [1]

To measure the performance of the following election algorithm the total network bandwidth utilization will be considered. [1]

In the following, we will analyze a collection of voting algorithms. In conclusion of the analysis, we will pick one algorithm which fits to our project requirements and promises the best performance.

3.2.1 Chang and Roberts Algorithm:

The algorithm of Chang and Roberts [1979] (CR algorithm) is suitable for a collection of servers arranged in a logical ring. Each participant has a communication channel to the next participant in the ring and all messages are sent clockwise around the ring. We assume that no failures occur, and that the system is asynchronous. The goal of this algorithm is to elect the leader, which is the server with the largest identifier.[1]

The algorithm can be separated in two parts, first:

1. Initially, every server is marked as a “non-participant” in an election.
2. Any server can begin an election by sending an election message containing its unique identifier and sends the message in a clockwise direction to its neighbour.
    - SE(pid, isLeader), with pid = the participants ID which is sent
3. Every time a server sends or forwards an election message, the server also marks itself as a “participant”.
4. When a server receives an election message, it compares the identifier in the message with its own unique identifier. Four different possibilities can occur.

    4.1 Firstly, if the identifier in the election message is larger, the election message is forwarded.
    - pid > mid, with mid = the servers own unique ID → SE(pid, False)

    4.2 Secondly, if the identifier in the election message is smaller and the server is not yet marked as a participant of an election, the server replaces the identifier with its own unique identifier and sends an updated election message.
    - State = “non-participant” AND pid < mid → SE(mid, False)

    4.3 Thirdly, if the identifier in the election message is smaller, but the server is already a participant of an election, the election message is discarded.
    - State = “participant” AND pid < mid

    4.4 And fourthly, if the identifier in the election message is the same as the unique identifier of the server, that server starts acting as a leader. [1]
    - pid = mid

The second part of the algorithm starts, when a server starts acting as a leader:

1. The leader marks itself as non-participant and sends an elected message with its election and identifier in a clockwise direction to its neighbour.
    - SE(pid; True)
2. When a server receives an elected message, it marks itself as non-participant, records the elected identifier and forwards the elected message unchanged.
3. When the leader receives its own elected message, it discards that message and the election is over.
4. The primary server informs the front end server about his election by a leader message LE.
    - LE(pid)

[1]

The architecture of the CR algorithm:

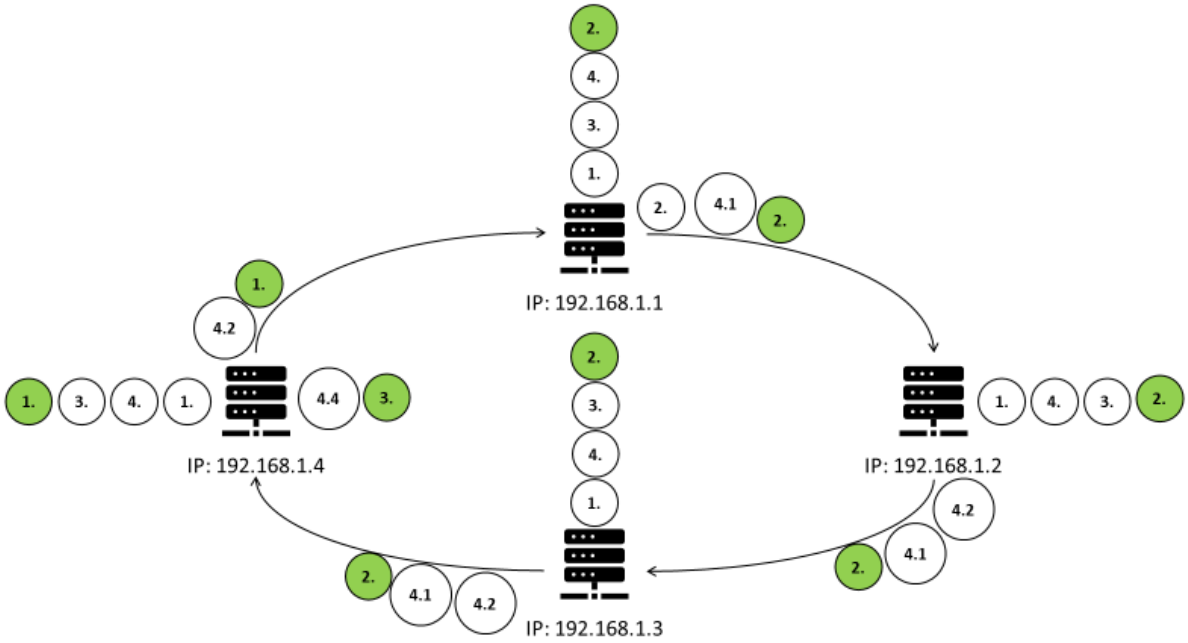


Figure 3: Chang and Roberts algorithm - just one server starts an election

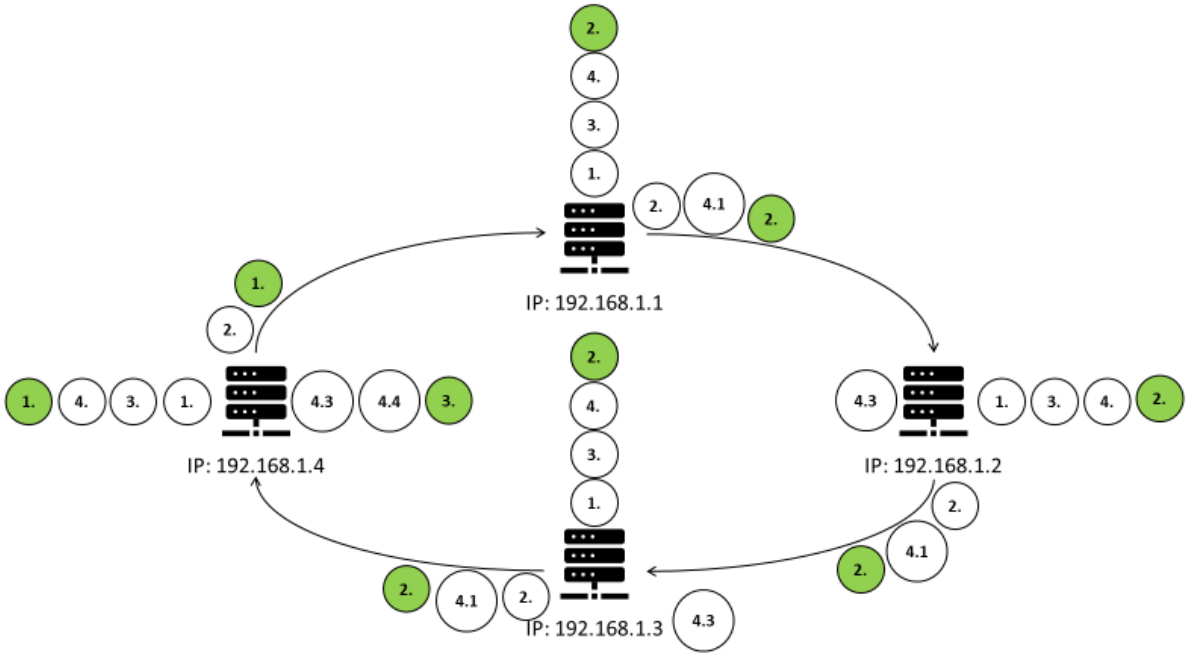


Figure 4: Chang and Roberts algorithm - every server starts an election simultaneously

The Chang and Roberts algorithm respects safety, as a server will receive an elected message with its own unique identifier only if his identifier is greater than others', and only when all servers agree on the same identifier. The algorithm also respects liveness. “Participant" and "non-participant" states are used so that when multiple servers start an election at roughly the same time, only one single leader will be announced. [3]

If only a single server starts an election, the algorithm requires $3n - 1$ sequential messages, in the worst case. [1] If every server starts the election, in a worst-case scenario also $3n - 1$ [1] has to be sent during the election process until a leader is elected.

In the following we will point out the differences to other algorithms and explain why we decided to use the Chang and Roberts algorithm for our project.

Hirschberg-Sinclair algorithm

The Hirschberg-Sinclair algorithms (HS algorithm) election messages are more complex compared to the CR algorithm and takes more time. Measuring the total network bandwidth utilization, we are regarding the worst case of the election comparing the two algorithms. We compared the case if every participant of the system is starting the election equally. The number of messages for the worst case of the HS algorithm can be calculated by $\left(\frac{\log(n)}{\log(2)} + 1\right) * 2n + n * \frac{\log(n)}{\log(2)}$ [1] with n as the number of participants. The number of messages for the worst case of the CR algorithm can be calculated by $3n - 1$ with n as the number of participants. Regarding the two equations, the CR algorithm has to send less messages. Even though the HS algorithm also respects safety as well as liveness, regarding our project, the CR algorithm is more suitable. That is because the total network bandwidth utilization is lower with the CR algorithm. Furthermore, the election message is less complex with the CR algorithm and no reply message is required.

Bully Algorithm

Regarding the requirements of the total network bandwidth utilization the CR algorithm is more suitable for our case. If one server starts an election, it takes n^2 messages in the worst case to elect a leader with the bully-algorithm. [1] By way of comparison, the CR algorithm requires less messages in the worst case, because its total network bandwidth utilization is $3n - 1$. The bully algorithm sends less messages for $0 > n > 3$ but the CR algorithm in a range of $3 \leq n < \infty$. For a system of three or more participants, the CR algorithm is more suitable regarding the total network bandwidth utilization.

Furthermore, the bully algorithm is not guaranteed to meet the safety condition as two servers can announce themselves as the leader concurrently. Unfortunately, there are no guarantees in message delivery order and these could lead to different conclusions on which of the servers if the leader. [1]

Despite the fact that the CR algorithm tolerates no failures, with a reliable failure detector it is in principle possible to constitute the ring when a participant of the system crashes. For our project to understand the properties of election algorithms in general and with a small number of servers, a ring-based algorithm is useful. [1]

3.2.2 Our Solution and the technical restrictions

As we have mentioned above, for our project we will use the CR-algorithm for the leader election.

Ring formation:

First, we have to implement an asynchronous uniform non-anonymous ring. Therefore, we use the participants IP address as unique identifier. This has the consequence that the participant with the highest IP address also later will be the elected leader. The ring is formed by sorting the list of the IP addresses. See **Fehler! Verweisquelle konnte nicht gefunden werden.** in Chapter 3.

[1] details on the derivation can be found in the appendix

The ring formation follows the following steps:

1. For the ring implementation, we assume that the participants of our system are all members of the same group of servers and have the same view of the group.
2. Afterwards, the participants sort themselves in ascending order of their IP addresses. Each participant knows its clockwise neighbour.
3. Each participant knows its counter-clockwise neighbour

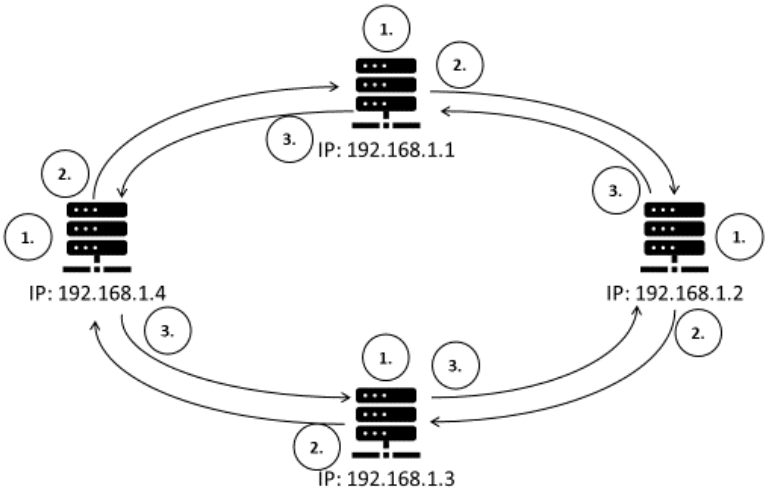


Figure 5: Ring formation

Election algorithm:

As we use the CR-algorithm, the link between the participants are unidirectional and every server can send the election message to its clockwise server only. The functionality of the CR algorithm has already been explained above, but there will be examples of its implementation in the architecture design section and in the code repository. With a CR leader election algorithm, we provide that we have at least and at most one leader. This one is the first back end server, which responds to client requests.

Failure detector:

As a failure detector which will provide system failures caused by crashed participants, we implement heartbeat messages. These are “messages sent at fixed time intervals to indicate that the sender is alive” [1]. When a server does not receive a heartbeat from its neighbouring participant, an update on the view of the system is sent out and if necessary, a new leader election starts. Even in the case of multiple simultaneous failures, this procedure is able to keep the system working.

The mechanism of the heartbeat messages can be separated into two parts, firstly the distribution of the heartbeat message:

1. Every second, the leader creates a heartbeat message (HB) sends it unidirectional and clockwise through the ring. It records the time it has send the message.
    - HB(heartbeat GUID)
2. Every time a backup server receives a heartbeat message, it records the time and forwards the heartbeat message.
3. When the primary server receives its own heartbeat message, it records that the ring is complete, and all backup servers are available.

Secondly, the reaction on crashes in the system:

1. If the interval since the last heartbeat exceeds a timeout threshold of three seconds, the detecting server starts a new ring formation by sending a failure message (FF) via multicast to all other participants. [1] This failure message includes the ID of its counter-clockwise neighbour. If the crashed participant was the leader, it also starts a new leader election.
    - FF(IP address of failed node [counter-clockwise neighbour])
2. When a server receives a failure message, it updates its group view on the system.

The architecture of the failure detector:

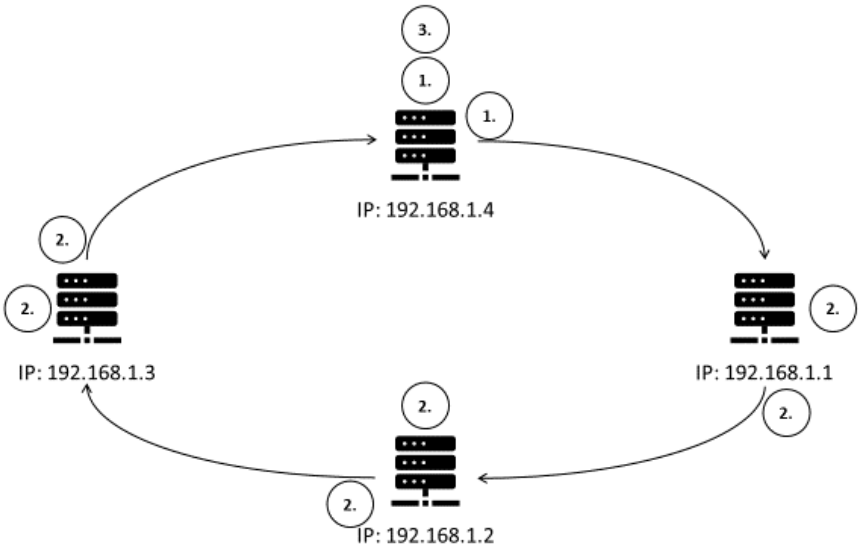


Figure 6: Failure detector: heartbeat messages

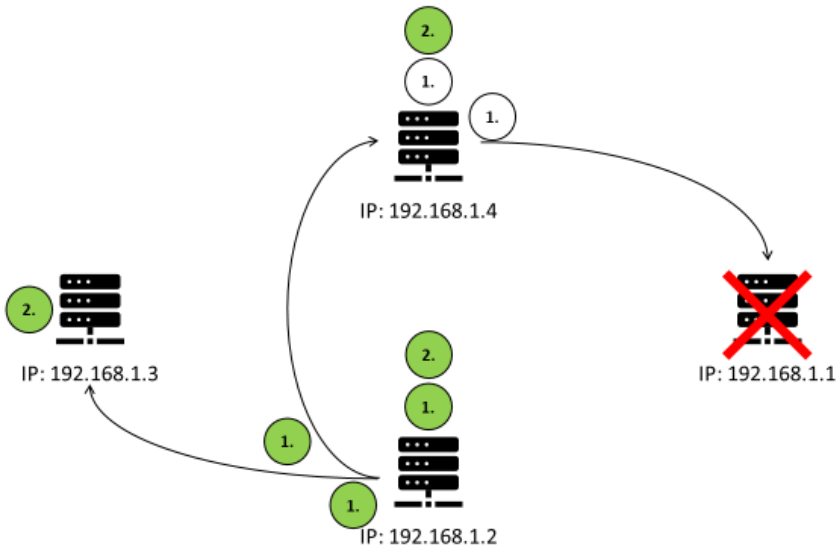


Figure 7: Crash tolerance: heartbeat messages

Problems related to the incorrect transmission of data will not be a focus of this project but will mentioned in the discussion and future work chapters.

3.3 Logical time and reliable multicast with causal ordering

3.3.1 Logical time

As we cannot synchronize clocks perfectly across a distributed system and the physical time does not work to order events that occur at different servers, we have to implement scheme which is similar to physical causality. Therefore, we implement a logical clock to provide how a local server updates its own clock if an event like a new quote is posted. Also, we have to determine how a local server update its clock when it receives a message from another participant for example a new comment. To provide causality consistent ordering we introduce vector clocks. The vector clocks are used to causally order messages. Each server maintains a vector with a sequence number which tells us how the partial order happened. Basically, each counter represents the number of messages received from each of the other participants. So, if a client sends a message the sender increments its own counter after the message was acknowledged and attaches its vector clock. Also, the vector time clock is suitable for our causal ordered multicast.

3.3.2 Causally reliable multicast

A reliable multicast is needed to provide the quotes to every slave of the replication system. In addition to the reliable multicast we add up the causal ordering. A causally ordered multicast is needed to guarantee a relationship between posting and reply function.

In order to implement the vector timestamp in our project, we modified the vector presentation slightly. Instead of representing the vector with unnamed values like [0,0,0], we added the IP address of each participant as identifier for the corresponding position which looks then like [A:0,B:0,C:0].

How does it work?

1. All processes start with a vector clock timestamp of [0,0,0]
2. A user is sending a quote to our system, therefor we multicast the message m to group g. As a result the process will increment the timestamp by 1 which is included in the message.
3. Before the message is delivered, the messages that were previously received are held in a hold back queue. These messages have to be delivered first, before our mentioned user message gets delivered. Each received message will also increment the timestamp by 1.
4. If the queue before the message m is empty, it will be processed and delivered. The process delivers message m and updates the timestamp by increasing it by 1.

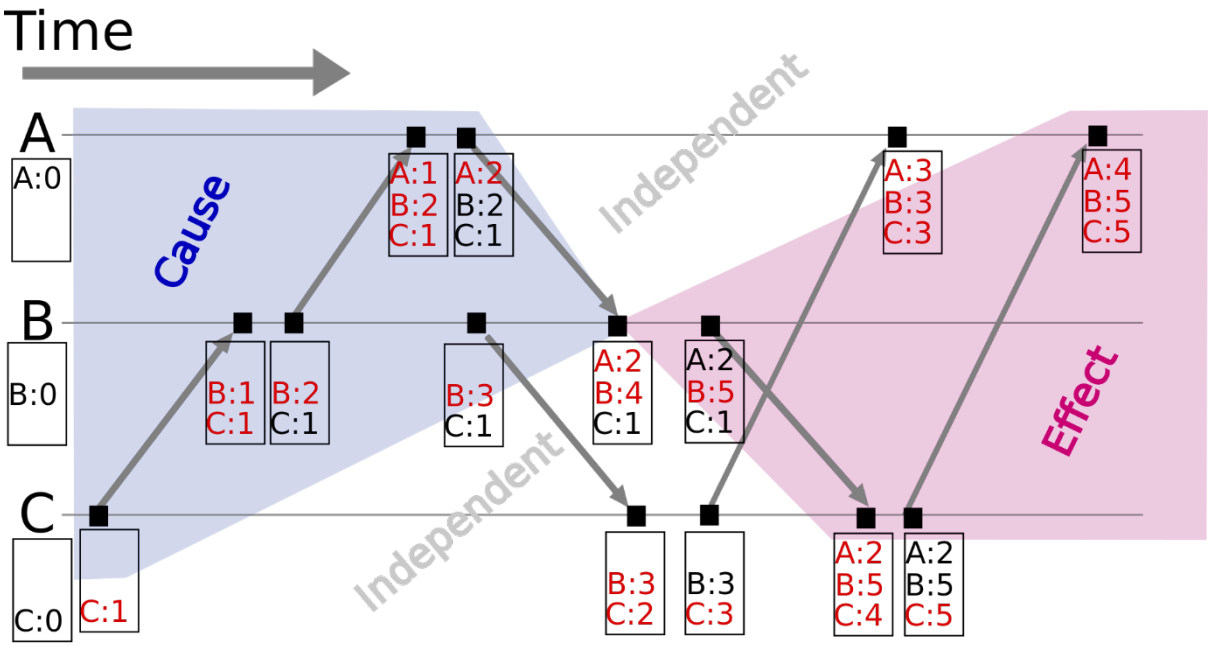


Figure 8: Example of a system of vector clocks [4]

Why we did not choose basic multicast:

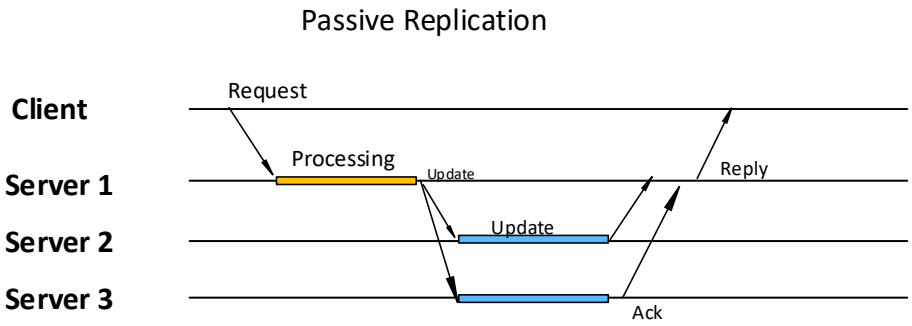
➔ Ack implosion → multicast server buffers fill rapidly → as a result we have to retransmit messages which further results to more acks and more waste of network bandwidth.

With the reliable multicast we keep a history of messages for at-most-once delivery and everyone repeats the multicast upon a receipt of a message for agreement & validity. So, we achieve:

- "Agreement: If a correct server multicasts message m then it will eventually deliver m.
- Validity: If a correct server delivers message m, then all other correct servers in group(m) will eventually deliver m."[1]

3.4 Passive Replication

3.4.1 How does it work? – passive replication



In the passive, also known as primary backup model of replication, we have one designated primary replication manager who is responsible to manage the communication between client requests and the replication of data to the backup servers. In our architecture there is at most one primary server who deals with the incoming requests of the clients which is represented by the leader. The backup servers ignore the client requests permanently until they get elected as primary.

1. The client sends a request to the front end to get to know the ID of the primary server
2. The front end sends the primary ID to the client
3. Then the client sends a content request to the primary only.
4. The primary takes each request atomically, in the order which it receives it and checks the unique identifier.
    5. In case of receiving an update, the primary manager updates the other copies and sends updates to the backup servers
    6. The backup servers save the received copies.
    7. If the request has been executed, the primary sends a response to the client again.
    5. In case the primary has already received the request once, it ignores this redundant message.

The architecture of the passive replication:

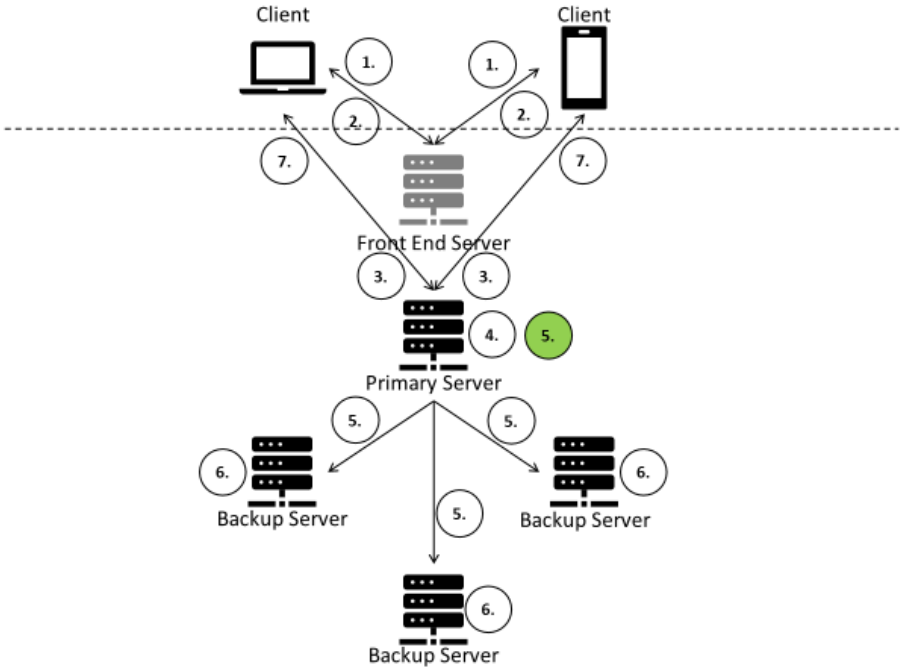


Figure 8: Passive Replication

Since we only have one active primary replica manager and the group members want to continue using the service, we need to make the service fault tolerant regarding the primary server crash. Besides a new leader election, the backup servers have at this point no knowledge of the primary server's state. For example, there are periods where no quotes are posted, so the backup servers assume that the primary server is probably crashed.

To prevent this from happening, we also use the heartbeat messages which we already explained in chapter two. The frequent interval of one second ensures that a failure of the primary is quickly detected, and a new leader election is rapidly conducted. As a result of the new primary leader election, clients cannot identify the difference between a service which is based on the primary server or the service obtained by replicated data.

3.4.1.1 Correctness criteria's

Linearizable

Obviously the linearizability is achieved through a correct primary which sequences all the operations. In case of primary failure, the system can preserve linearizability with a new primary election, where a backup server gets promoted to the new primary server. The new primary must take over exactly where the last primary left off.

Crashes:

If the primary crashes, backups will receive new view with primary missing and initiates a ring election. A new primary gets elected.

3.4.1.2 Advantages

- All operations passes through a primary that linearize operations. Each request is atomically taken in order. Therefore, we do not need a coordination extension of FIFO etc.
- Works even if execution is non-deterministic.
- If view synchrony is given, no consultation with the backups is needed, because they all have processed the same set of messages.

3.4.1.3 Disadvantages

- Delivering state change can be costly.
- View synchrony and leader election could be expensive.
- Failure response is delayed (in comparison to active replication).

3.5 Why did we choose passive replication?

We choose the passive replication because we already follow a casually ordered multicasting approach. In case of the totally ordered multicast, we would have probably a big hold-back queue due to many client requests. Our goal is to achieve high availability and good performance. Therefore, we also implement a variation of the stated model above. Another disadvantage of the active replication is that the requirement of replicated objects has to be deterministic, which results in redundant processing. Whereas the passive replication does not waste resource and permits non-deterministic operations. But we have to deal with an increased latency of an invocation. Also, we have to cope with the update state and its distribution to the backup servers.

Furthermore, the architecture given by the passive replication enables us the use of a primary copy replication. Primary copy replication is used in context of transactions, in our case all client requests are directed to a single primary replica manager. As a result, for primary copy replication, we control concurrency through the primary.

4. Implementation

Our project is written with the language of Python. Besides the provided script, we have read other internet sources and books. Intellectual property was respected according to the rules of scientific work and is therefore marked accordingly. As main source we used the book "*Distributed Systems – Concepts and Design*" written by George Coulouris, Jean Dollimore and Tim Kindberg which is quoted several times. The code for our distributed system can be accessed on the following Github link at 4.1. Code Repository.

Besides our distributed system we added a front end server as web server. Therefore, the client requests over the front end the index.html, gets access to the Webpage and requests the back end server for the content socket connection at /leader. The logic is implemented on client side with javascript. As a result the client gets a json file back with the IP address of the back end server. Then he connects via websocket to the back end server at "ws://[ip-address]:10013/".

The UML class diagram which can be found in the appendix or at https://git.io/JJTrT, illustrates the different classes with their characteristics and their correlation to each other.

4.1 Code Repository

The code repository can be found at https://github.com/davvve/DS_SS20_Group16_localshit.

5. Discussion and conclusion

As our first self-developed distributed system, we build an application which allows his users to share quotes with others and react on them. We used the UDP Datagram to build up the communication between the client and our primary server with usage of sockets. The primary server is the participant with the highest IP address, which is elected as the leader. This leader election is possible after a ring formation and with usage of the Chang and Roberts algorithm. To guarantee crash fault tolerance, we implemented heartbeat messages, which the leader is sending through the ring to check if every server is still running. We implemented causal ordered multicast. The causal ordering is used to order the comment function. In addition to that, we choose a vector clock to ensure synchronized time. The content will be processed by the primary server and also replicated to the backup servers.

Regarding the requirements of a distributed system we mentioned in the introduction, we were able to fulfill the most parts. Even though we used Python for our implementation, every other programming languages and algorithms can also be used to establish similar programs, which makes our system heterogenic. Our program is also an open system because there are several possibilities to access it. The clients can use a device of their choice. Our system is designed so that external servers can easily be included in the program. Next to that, the data is structured hierarchically to get the best access time. This makes our program scalable. Two other requirements, which we were able to fulfill are concurrency and transparency. Regarding concurrency, our system can handle several requests simultaneously, as we use causal ordering of the incoming messages. Within the system there is complete transparency of each processes and data. Even in an external view, we can guarantee full transparency in access and mobility. Another requirement, which we could fulfill, is crash tolerance. The data from the client is replicated by our backup servers. A replacement of crashed servers is easily possible, by any of the other backup servers, because every server has the same data knowledge. As a result, we reduce the risk of data loss. Each participant can crash independently without causing a crash of the whole system.

But we are not resistant to byzantine faults. This means, that content could be changed by faulty transmissions between the client and the primary server, as well as between the servers.

Our distributed system does not fulfill every security requirement, which we named in the problem statement. This is an issue that can be fixed in further steps of the application development.

For example, to verify correctness and completeness of the messages, we could introduce the concept of checksums. A checksum algorithm produces a unique value of the message. This enables the receiver also to run the function over the message and to compare the values.

Another problem regarding crash tolerance of the complete system occurs at the front end server. This is a bottleneck between the client and the backend. To fix this, we would need multiple frontend servers which operate simultaneously.

5.1 Future Work

In a further step or stadium of our project, we can face the challenge to fix the security and byzantine fault tolerance issues. To fulfill the security requirements of distributed systems, could implement digital cryptography which provides the basis for most computer security mechanisms. [1] To solve the problem with possible byzantine faults, we could think about additional messages between the replication manager and the backup servers, to make sure everyone has the same data content and no transmission failures occur. But we would also have to implement reading rights to the backup servers, that they could review the primary server.

In addition to the fulfilment of the requirements above, we can think about uploading images instead of text or a vicinity-based service, so that the content is only visible for users in a limited area of approximately 10 kilometers around the person who uploaded the content. But for the first implementation we focused on the technical topics of the lecture. These two ideas require high amount of client-based technology like GUI programming or location-based services.

6. Bibliography

[1] G. F. Coulouris, J. Dollimore, T. Kindberg, G. Blair, A. K. Bhattacharjee, and S. Mukherjee, *Distributed systems : concepts and design*. 2012.

[2] F. A. Dalwigk, “UDP vs. TCP - Algorithmen verstehen,” 2019. [Online]. Available: https://www.cybersicherheit.guru/tcp-vs-udp/. [Accessed: 07-Jun-2020].

[3] “Chang and Roberts algorithm,” 2019. [Online]. Available: https://en.wikipedia.org/wiki/Chang_and_Roberts_algorithm. [Accessed: 07-Jun-2020].

[4] “Vector clock,” 2020. [Online]. Available: https://en.wikipedia.org/wiki/Vector_clock#/media/File:Vector_Clock.svg. [Accessed: 01-Jul-2020].

7. Appendix

7.1 Proof of total network bandwidth utilization (n)

7.1.1 Hirschberg-Sinclair algorithm worst case with 8 participants:

Current phase (k)	Number of messages (n)	Type of message	Check if $d = 2^k$
k=0	n*2	Election	$2^0 = 1$
k=0	n	Reply	$2^0 = 1$
k=1	n*2	Election	$2^2 = 2$
k=1	n	Reply	$2^1 = 2$
k=2	n*2	Election	$2^2 = 4$
k=2	n	Reply	$2^2 = 4$
k=3	n*2	Election	$2^3 = 8$
In general:			
k(max)	n*2	Election	$2^k = n$

Getting k: $2^k = n \rightarrow \frac{\log(n)}{\log(2)} = k$

Regarding the election messages: $(k+1) * 2n$ | (k=0 must be considered!)

Regarding the Reply messages: $k * n$ | (k(max) must not be considered!)

➔ $\left(\frac{\log(n)}{\log(2)} + 1\right) * 2n + \frac{\log(n)}{\log(2)} * n$

7.1.2 Chang and Roberts Algorithm worst case with 8 participants:

Worst Case: If the clockwise neighbour of the primary is starting an election. Or every server starts an election simultaneously.

The clockwise neighbour of the primary is starting an election:

Number of iterations	Number of messages(n)
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8-15	8
16-23	8

For n-1 iterations: $n - 1$

For n to 2n-1 iterations: n

For the 2n to 3n-1 iterations: n

➔ $3n - 1$

Every server starts an election simultaneously:

Number of iterations	Number of messages (n)
1	8
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9-16	8

* Note: the servers are always ordered in ascending order of IP addresses due to the ring formation. Therefore, all voting messages except the one of the later leader are already eliminated in the second iteration.

For n iterations: $n + n - 1$

For the n+1 to 2n iterations: n

➔ $3n - 1$

7.2 UML Class Diagram